AFRL-RH-WP-TP-2008-0009

Implementing HCI Design Patterns as Widget/Templates for GUI Builders

Donald West

Consortium Research Fellows Program
Washington D.C.

Vincent Schmidt

Air Force Research Laboratory
Cognitive Systems Branch

March 2008

Interim Report

Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7022

NOTICE

AFRL-RH-WP-TP-2008-0009

//SIGNED//
VINCENT A. SCHMIDT
Work Unit Manager
Cognitive Systems Branch

//SIGNED//
DANIEL G. GODDARD
Chief, Warfighter Interface Division
Human Effectiveness Directorate
Air Force Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
March 2008	Conference Proceedings	

4.TITLE AND SUBTITLE

Implementing HCI Design Patterns as Widget/Templates for GUI Builders

5a. CONTRACT NUMBER
FA8650-04-D-6405

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62202F

6. AUTHOR(S)

[1]Donald West, [2]Vincent Schmidt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
7184HEX6

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

[1]Consortium Research Fellows Program
Washington D.C.

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

[2]Air Force Materiel Command
Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7022

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RHCS

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-RH-WP-TP-2008-0009

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

2008 International Conference on Software Engineering Research and Practice (SERP'08), Las Vegas, NV, July 14-18, 2008. 88th ABW/PA cleared on 02 April, 2008, WPAFB-08-2516

14. ABSTRACT

During the development of a software system, something is lost in translation of the Human-Computer Interface (HCI) between human factors engineer's analysis and the software developer's implementation. Since the developer touches the product last, the human factors engineer's contribution is frequently lost. Graphical User Interface Design Patterns (UIDP) are templates representing commonly used graphical visualizations for addressing certain HCI issues. These patterns include substantial contributions from human factors professionals. Using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. This paper introduces the concept of the UIDP and describes how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user's work environment.

15. SUBJECT TERMS
Design Patterns, GUI, HCI, Widget

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Vincent A. Schmidt
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	10	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

THIS PAGE LEFT INTENTIONALLY BLANK

Implementing HCI Design Patterns as Widget/Templates for GUI Builders

Donald West
Consortium Research Fellows Program
Washington, D.C, USA

Vincent Schmidt
Wright-Patterson AFB, AFRL/RHCS
Dayton, OH, USA

Abstract - *During the development of a software system, something is lost in translation of the Human-Computer Interface (HCI) between human factors engineer's analysis and the software developer's implementation. Since the developer touches the product last, the human factors engineer's contribution is frequently lost. Graphical User Interface Design Patterns (UIDP) are templates representing commonly used graphical visualizations for addressing certain HCI issues. These patterns include substantial contributions from human factors professionals. Using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. This paper introduces the concept of the UIDP and describes how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user's work environment.*

Keywords: design patterns, GUI, HCI, Widget

1.0 Introduction

Early software benefited from the simple use of abstract data types (ADTs) such as stacks, queues, and linked lists. Even with these straightforward ideas that enhanced the coding for more advanced machines having improved graphics and processing power, programmers still struggled to find ways to reliably describe concepts frequently adopted within the core of the code. The 1994 "Gang of Four" book *Design Patterns: Elements of Reusable Object Oriented Software* [1] took the software development community by storm with its popular and accurate description of frequently used software design patterns. Using the patterns described in the book (and later, other patterns identified by the community at large), programmers could code reusable components with a degree of confidence that the models were complete and descriptive.

The design pattern concept is not new. Christopher Alexander is frequently cited as the inspiration of documented patterns with his 1979 book *The Timeless Way of Building* that describes patterns of architectural design [2]. Even if documented ADTs are considered as a form of early design patterns, there is no argument that the Gang of Four book significantly impacted the state of the art. Along the same lines, Jenifer Tidwell introduced a book describing design patterns as applied to user interface design [3], in which the first chapter is devoted to describing the relationship a user interface should have with the end user.

Human Factors Engineers (HFEs) specialize in, among other areas, the design of excellent user interfaces targeted towards the end user. The traditional approach to building computing systems is for the HFE to generate conceptual interfaces, and then pass these designs along to the programmers for implementation. Unfortunately, inadequate specification and gross miscommunication is the norm, so the well-designed interfaces are frequently generated in such a way that many of the elements intended to contribute to a well-designed interface are left out of the final implementation. The result is a new software system that does not optimally support the end-user's work environment.

One solution to this problem is to have a User Interface Design Pattern (UIDP) library that goes beyond the mere description or specification of Graphical User Interface patterns (GUI). (With the exception of ADT libraries, most design pattern work currently ends with the description of the patterns and small snippets of sample code. This leaves implementation details up to the programmer, introducing the strong potential for misrepresentation of the final GUI elements.) Such a library would include templates that can be parameterized to generate a "90% solution" to specific GUI elements in the design. Each one of these templates would be vetted by HFE experts, and their use would guarantee that human factors components are correctly represented graphically to the end user.

Our contribution to this solution is to provide these library components as "widgets" in a popular drag-and-drop GUI builder. Programmers could drag these human-factored UIDP elements into the applications being built, and an integrated wizard would guide the programmer through a series of tailored queries to

parameterize the template. The code generated by the GUI builder would include not only the code necessary to design the GUI's standard widgets (pull-down menus and button elements), but also code to generate the specialized UIDP elements. This will result in an application where human factors designs are not lost due to implementation miscommunications. Furthermore, the introduction of Human Factors into the standardization of design patterns encourages coders to use these pre-designed elements, since these elements will be more user-friendly and less prone to coding error. The details of our progress to date and anticipated future direction are documented here.

2.0 User Interface Design Patterns

Good human factors dictates that data be portrayed in a manner consistent with the work being performed, such that those accessing the information be able to quickly and easily read and understand the situation. There are common methods for visualizing certain types of data, where minor alterations in the visual display can be performed to suit a wide variety of situations and software applications. These visualizations are called User Interface Design Patterns, and they represent a class of repeatable, general solutions to commonly occurring end-user display needs.

As new user-interface design patterns are discovered, they can be documented in a design pattern library. This library can serve as a foundation to all user interface creation, much as Erich Gamma's [1] documentation of elements did for the creation of object-oriented software. A good pattern library would include a list of the patterns, a description and formal specification for each pattern, sample pattern usage code, and a reusable template.

Pattern software objects can be made with good human factors design already implemented, effectively representing a "90% solution" to a programmer's visualization requirements. The key benefit to such patterns is that high-quality code with built-in human factors design is available to the coder to support rapid software implementation. The resulting application will be well-suited to the user, meeting good human factors design criteria. (Coders are not traditionally trained in human factors sciences, so many of their interfaces poorly support the user or the work environment.)

The subsections that follow describe the user interface design pattern concepts by way of example. The Map and Timeline patterns are used within our own organization as custom-coded components, but we are working to show how they can be used by a wider audience using a general software development framework.

2.1 Map Pattern

The Map pattern is useful for displaying spatial information as layers of imagery, symbols, and corresponding text. As the name suggests, this pattern is found primarily in maps and mapping software. Since geographic paper and electronic maps are commonly used, most people are already familiar with the map concept.

In general, maps are frequently used for driving directions and road map scenarios. However, this pattern could also be used in a wide variety of other scenarios: creating a weather map for a new area, mapping underground tunnels, visualizing something as small as a microchip and schematic diagrams, or displaying something as large as the night sky.

Figure 1. Google Maps Example [4]

The map pattern can be implemented as a widget that is a collection of layered images and corresponding text, along with extra control options and map legend data. The example shown in Figure 1 could easily be created with two layers. The first would be a base layer consisting of a picture of the area. The second would be a drawn image of roads that coincides with the base layer's picture.

Each layer could have a collection of related attributes and controls, such as:

- Image
- Opacity
- Vertical Pan
- Horizontal Pan
- Vertical Wrap
- Horizontal Wrap
- Zoom qualities
- Zoom rate
- Toggle on/off

These characteristics might be explicitly coded into the map implementation by the programmer, or they may be

configurable in real time by the user at the user-interface level.

As an additional example, the map pattern could portray a floor plan for an entire multi-floor building (Figure 2). The pattern is instantiated by adding a layer for each floor and controlling layer opacity. Clearly, the map is a good pattern because of its ability to be reused in a multitude of situations.

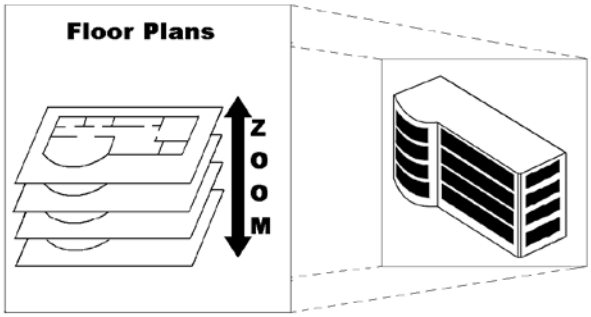

Figure 2. Floor Plan Concept

The map pattern could be implemented as a template that is instantiated within a GUI builder. When the programmer adds the map widget to the display, a wizard guides the GUI designer through a series of decisions to fill out the code template. The map's layer concept is represented by providing the GUI designer an opportunity to continue to add new layers until all desired layers have been described. For each layer the user adds, the wizard provides a series of options and attributes that can be selected for this layer, and indicates how the layer relates to other layers. (For example, should opacity be defined by the programmer, or should the end-user be able to change opacity dynamically for this layer? Should this layer be registered to pan and zoom independently, or tied explicitly to the corresponding values of other specified layers?) The wizard also allows the layers to be prioritized, stacked, and placed as a cohesive unit. While there are only a few attributes, adjusting them can lead to an optimized "map" for a vast number of different uses.

Suppose there is a need to display weather information on the map shown in Figure 1. By adding a layer containing a dynamic image of a Doppler readout, giving this new layer a high priority (or top location on a visible stack), and changing it's opacity to 50%, clouds and precipitation would be shown along with the referenced geographic data. This layer could also be deselected so it is not displayed. Clearly, the layers need to be able to display both static and dynamic information, as well as textual and symbolic data. The map pattern template must provide a mechanism for these displays.

Some characteristics certainly need to be determined by the programmer. The map pattern template should provide code providing these decision points and implementing the capabilities required. Navigation and perspective are prime examples. Additional navigation options may be necessary when an image becomes too large for the screen. This is where the pan, zoom, and horizontal and vertical wrap attributes come into play. A programmer may add a slider for measurements that do not pan with the other layers (such as distance or angle overlays and other "heads-up display" information). Similarly, a programmer may require layers to pan at different rates (perhaps to approximate differences in distance between successive layers; a rotatable first person perspective and pan is similar to rotating the point of view, so closer objects will pan more slowly than farther objects).

Another interesting map pattern usage is zoom capability. The programmer will need to decide if the user has a requirement to zoom into the image of layers, or zoom from one layer to another, or both. If the programmer makes interactive blueprints for an office building as shown in Figure 2, then it may be desired for the zoom function to go up or down one floor, whereas the Google Maps application zooms into the layer images until the image quality cannot be maintained, then switches to another layer.

The map pattern is common and versatile, and its many implementation characteristics make it a clear candidate for incorporation into a user interface design pattern library. This discussion treats the map pattern as a two-dimensional pattern; it is unclear at this time if it is a good idea to directly extend the map pattern into a 3-D (or other multi-dimensional) pattern, or if such usages should eventually be implemented independently.

2.2 Timeline Pattern

Timelines are an integral part of today's professional world, whether project management, corporate planning, military missions, or even just trying to reserve room C-3 down the hall for a conference next Tuesday. This pattern is hidden everywhere in our daily lives as well, and can be seen whenever we look at a calendar or make an entry in our PDAs. At its simplest, a timeline pattern is nothing more than a standard way of visualizing time along with constraints and obligations. Incorporation of human factors considerations into the implementation adds tremendous value to the timeline pattern as a software component.

The timeline pattern shows linear time using a

collection of simple numberline-like graphs: lines and points correspond to important moments or time spans. Dependent timelines can be used in conjunction with each other to account for objects, people, or events that mutually constrain the same time periods. Consider a scenario where you want to set up an office recognition party for Jim during business hours. The facility has three conference rooms suitable for the event, but they are intermittently reserved for meetings. The party is expected to last an hour and a half. Bob and Dave, Jim's best friends, must also be able to attend. Jim, Bob, and Dave all have various daily commitments, and Jim has to oversee the activation of the Jumbo-tron at 3 p.m. each day.

There are several ways someone could deal with scheduling Jim's party with this information. They could try to "think it through" in their head, but this grows increasingly difficult with each new constraint or additional piece of information. The data could be placed in an electronic spreadsheet, but this format is often sloppy, complex, prone to error, and could take a long time to analyze when cross-referencing all the data. This type of data could be easily visualized with a timeline, such as in the example of Figure 3.

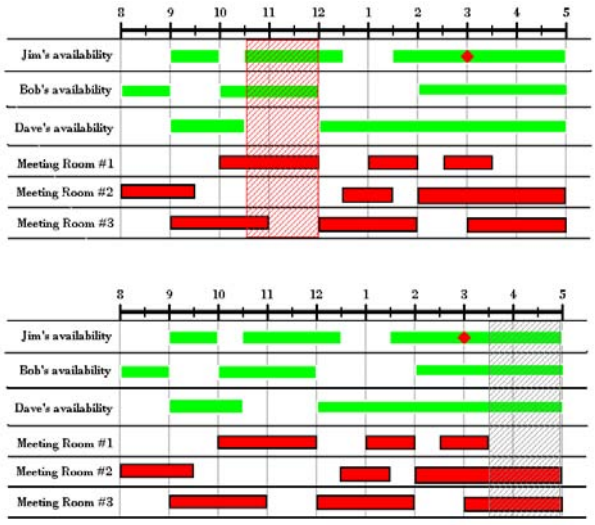

Figure 3. Jim's Party

The timeline formats all the data into an easy-to-comprehend view that doesn't require manually incorporating constraint cross-references as in an Excel spreadsheet.

The basis of this pattern is a simple linear reference timeline, shown by the notched line labeled from "8" to "5" in Figure 3. This base timeline will act as the point of reference for all related timelines. Each labeled row (shown below the reference timeline) contains a visual representation of the relationships and constraints of a specific activity to the reference timeline. These activities could also be related to (or constrained by) one-another.

Within these rows, there are three separate markers to identify important points or tracts of time:

1) **Lines**: These generally signify the primary information of an informative line.

2) **Points**: These signify an instantaneous action, restriction, or event.

3) **Backgrounds**: These are similar to lines, however they are placed behind lines and point to better use them in conjunction with the other two markers. Backgrounds are generally used to identify an acceptable period of performance for a desired action.

Lines, points and backgrounds can have constraints placed upon or between them among the timeline's rows. In Figure 3 the constraints can easily be seen; all people must be available, and at least one conference room must be available for an hour and a half period. As more informative lines are added and constraints grow more complicated, it will be necessary to let a computer keep track of the constraints. This is where a Timeline pattern becomes even more useful.

Just as with the Map pattern, the Timeline pattern involves pre-implemented human factors design. By progressing through a wizard, a programmer can easily create a base timeline, informative corresponding rows, and constraints amongst them to match any scheduling or planning needs.

Further aspects of the Timeline allows for tooltips. This enables detailed, relevant, contextual information of a line or object to be displayed by hovering over the object with the mouse. Tooltips provide easy access to contextual information without confusing the user and cluttering the screen when the information is not needed.

In addition to the look, feel, and functionality timeline features, a simulation mode can also be easily added. The simulation mode allows the user to graphically move lines and points (time constraints) without changing the actual configuration of the timelines. This capability allows the user to rearrange obligations and experiment with timeline configurations without impacting the actual data. The programmer is responsible for specifying the lines, points, and variables that can be moved, as well defining any restriction upon moving these objects. The simulation

mode also implements warning to indicate broken relationship constraints.

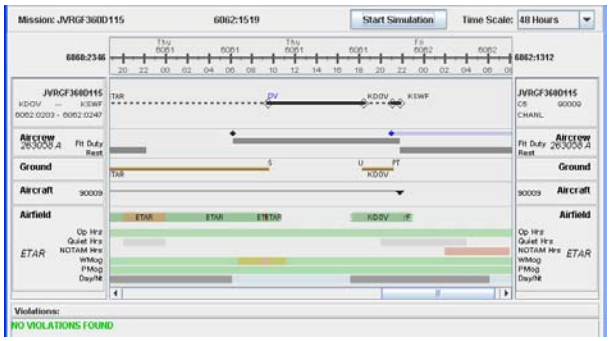

Figure 4. Timeline Tool Application

A Timeline pattern can be used to represent immense and complicated temporal relationships for extremely complicated scheduling tasks. The United States Air Force intends to use timeline patterns for important coordination of aircraft, flights, and refueling. The timeline pattern described in this section is a parameterized template based on a custom-coded Timeline Tool application (see Figure 4) being developed within the Air Force Research Laboratory. Our goal is to implement the timeline pattern such that the Timeline Tool could be created using the Timeline pattern wizard.

3.0 Implementation Decisions

Having defined these design patterns, a method of implementing them must be found. Our desire is to implement UIDP objects as drag-and-drop widgets within an existing GUI builder framework.

Implementation platform decisions are largely guided by our selection of the target programming language (i.e. Java, C++, Python, Perl, etc.). The list of potential languages to be used narrowed by our needs: we must be able to create visual objects that are easily manipulated by the typical programmer. The most commonly known object oriented, visually effective languages include Java, C++, and Visual Basic.

Java was selected as the language of choice because:

1) Contemporary programmers are being trained primarily in Java.
2) Many projects used within our organization and by our customers are coded in Java.
3) Java code is platform independent, meaning it can be used on any computer architecture with a JVM.

Once the computing language is chosen, the next step is to choose an integrated development environment (IDE). Not only does an IDE provide us (the pattern implementers) an easy, consistent coding environment by highlighting Java syntax, properly indenting lines, providing real time syntax checking, offering debugging tools, and providing a compiler, it also serves as the baseline for the GUI builder.

The selected IDE must support a GUI builder for Java, be extensible, and be relatively inexpensive (if not free). An extensible GUI builder is required, as these are *visual* design patterns that we are attempting to create. Extensibility is vital towards allowing us to incorporate extensions for our new pattern widgets when they are created.

As long as the GUI builder is a technologically sound product, low GUI builder cost will encourage frequent use of our patterns within the community at large. The cost of the IDE is also a convenience for us as implementers. If the IDE is free, that also takes a burden off from the end-user as they will be able to use these widgets without dipping into their wallets, or filing corporate purchase requisitions.

Table 1. Ranked Compilation of Java GUI Builders for Eclipse [5]

Name	Source	Price	Extensible?	Comments
Eclipse Visual Editor Platform	http://www.eclipse.org/vep/WebContent/main.php	Free	Yes	MAC OSX not supported
Cloudgarden's Jigloo SWT/Swing GUI Builder	http://www.eclipse-plugins.info/eclipse/plugin_details.jsp?id=472	Free	Highly Customizable	Good reviews, but extensibility is not clearly defined.
SWT Designer	http://www.swt-designer.com	Commercial License - $299	Yes	Restricted trial version available
Eclipse GUI Builder 2.2	http://eclipseplugincentral.com/Web_Linkes-index-reg-viewlink-cid-783.html	Free	Extendable Libraries	Has potential
Jvider	http://www.jvider.com	Single License - $69	Unclear, but suspect no.	Only usable on outdated Eclipse version
V4ALL	http://v4all.sourceforge.net/index_start.html	Free	Poor Documentation, unknown.	Shoddy documentation. Seems unprofessional
Matisse for Eclipse	unknown	unknown	unknown	Could only find negative review articles. Couldn't even find a website for it.

The most commonly used IDEs that are suitable for our needs include Sun Microsystems' Netbeans and the Eclipse Foundation's Eclipse IDE. Both Netbeans and Eclipse are free, extensible, and commonly used in the Java coding world. Eclipse was chosen because many of our own staff and customers are already familiar with this environment.

Eclipse offers a Java programming environment with very customizable windows and views. On top of its default customizable design, a user can download or create extensions and attachments for nearly every aspect of the IDE. This extensibility allows us to search through a wide variety of GUI builder plug-ins for Eclipse. Using the same criteria as for the IDE we compiled a list of seven possible GUI Builders and ranked them according to cost and extensibility. Ranked from best to worst, they are shown in Table 1.

Visual Editor was chosen for its clear extensibility and free cost. Cloudgarden's Jigloo was a close second. Unfortunately, its extensibility could not be fully determined without downloading and testing the software. With our own software installation approval process to go through, and limited time, it seemed prudent to use Visual Editor, which is already available and approved for use. (SWT Designer and Jvider were not explicitly tested due to the additional costs. V4All and Matisse for Eclipse may be good GUI Builders, but not enough information was readily available, so they were also dismissed.)

4.0 Design Architecture

Implementation of the UIDP concept is currently in the design and initial implementation phase, using Visual Editor as a baseline. Visual Editor is an easy to use Eclipse-based drag and drop GUI Builder which keeps manual GUI coding to a minimum. The Eclipse Visual Editor Platform allows for round-trip engineering: visual modifications are reflected immediately in the code, and code changes are displayed visually. Visual Editor also contains the fundamental Java widgets for graphical user interfaces such as buttons, scrolls bars, text boxes, check boxes, and radio buttons.

Our goal is to use Visual Editor to create UIDP tools as widgets to be placed in its existing library of widgets (Figure 5). These tools can then be used in a drag and drop fashion along with the IDE's pre-existing widgets. With the addition of a wizard to these new widgets, a fully customized high level pattern could be implemented in a matter of minutes.

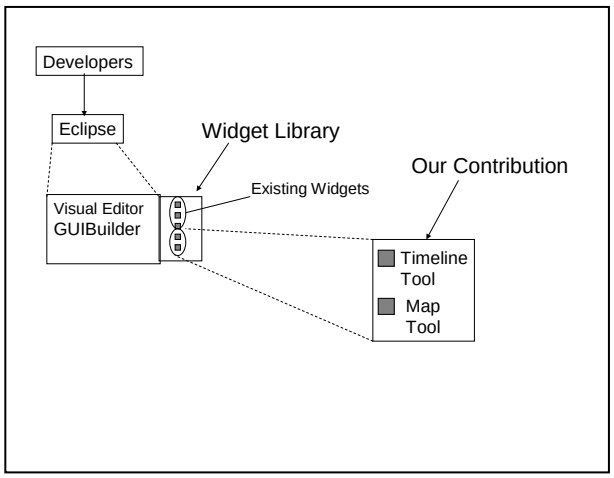

Figure 5. UIDP Implementation Architecture

Our rationale is that since GUI-based software is frequently built using GUI-builder tools, it seems reasonable to extend these drag-and-drop interfaces to include a collection of "super widgets" that implement the User Interface Design Pattern concepts. The coder would use these UIDP elements within the GUI builder just like the common widget set. UIDP elements themselves will be stored in a local library of coding templates. When the UIDP element is selected and placed on the display, a wizard will guide the programmer through a series of selections that instantiate the templates based on the desired characteristics.

The GUI builder will save the UIDP code along with all other generated code. Wizard selections should be saved (as XML, for example) so UIDP instantiation choices can be revised if needed. This information might be saved as an additional file, or it may be saved as comments within the generated code.

Implementation of the UIDP "widget set" within Visual Editor is module-based. The UIDP capability can be loaded into Eclipse as an additional module extension to the Visual Editor modules. We are currently concentrating on developing an understanding of how to interoperate with Visual Editor, including determining how to add new visual elements and how to generate code within this environment. Our final loadable module product is expected to be uploaded for use by the entire Visual Editor community.

All widgets within the UIDP module will already be vetted by human factors experts. These experts will be heavily involved in the design and evaluation of UIDP components. As coders who work closely with human factors engineers, we are somewhat uniquely qualified

to produce the UIDP elements for this library in such a way that the instantiated and generated library code results in visualizations with built-in human factors considerations that support the end-user's work environment and needs.

5.0 Conclusion

Efficient and effective software design is the grail sought by software professionals and technologists worldwide. Incremental steps are continually being made toward this objective: more accurate computing paradigms and information theory, specialized computing languages, and more capable and more complex software libraries. Together, these enable programmers to generate and maintain software systems faster, with likelihood of fewer bugs, and with greater flexibility than ever before.

A user-interface design pattern library is the next evolutionary step in software libraries. With design pattern tools such as the map tool and timeline tool, an easier method of implementing these documented patterns is formed, than with a small description of a pattern and a snippet of code. Instead, an interactive template will guide the creator to produce the desired user-interface with minimal errors, and a Human Factors Engineering component is already included in the code templates.

The implementation architecture is simply an effort to create a proof of concept for User Interface Design Patterns, and these preliminary implementation efforts are still subject to change. Even as this paper was being written, our research has uncovered a potential new method for pattern implementation, based on the Glade User Interface Builder. Glade seems to have the potential to build and add new widgets quickly and easily while allowing these additions to be language independent through the use of XML.

The development and documentation of User Interface Design Patterns has the potential to decrease build time, and increase product quality, reduce programming errors, and improve coding efficiency. Most of all, though, use of these patterns at the coding and design levels will improve the user experience, since these design patterns include a human factors component that supports the ability of the end-users to complete the work.

5.0 References

[1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1st ed. Reading, MA: Addison-Wesley, 1994.

[2] Alexander, Christopher. The Timeless Way of Building. 1st ed. Oxford: Oxford University Press, 1979.

[3] Tidwell, Jenifer. Designing Interfaces: Patterns for effective Interaction Design. 1st ed. Sebastopol, CA: O'Reilly Media Inc., 2005.

[4] "Google Maps." Google. 07 February 2008. Google. 7 Feb 2008 <http://maps.google.com/>.

[5] Stuart, Mitch. "Java GUI Builders." fullspan. 06 April 2005. fullspan. 7 Feb 2008 <http://www.fullspan.com/articles/java-gui-builders.html>.